Application layer thread migration in Java

Project report for Advanced Topics in PLT

Nikhil Sarda

ns2847@columbia.edu

1 Introduction

The continued validation of Moore's law has resulted in not just smaller and faster microprocessors, but also the development of storage devices having large capacities. The availability of such devices has resulted in a deluge of data being generated and gathered from various sources. One of the primary challenges of computer science today is the development of algorithms and systems that can take advantage of the availability of large and rich datasets.

However, making use of large datasets is easier said than done. Data produced as a result of scientific investigation typically resides in a single large data warehouse. Transferring all this information from one place to another is not always possible. For one, the existing bandwidth constraints on current network infrastructure might render this approach unfeasible. Besides physical limitations, there is also the issue with the law. Certain laws such as HIPAA (Health Insurance Portability and Accountability Act) prevent the sharing certain types of data without the express approval of the source. These laws essentially prohibit the free sharing of certain kinds of data over public channels. Another issue is that often times, data is propeitary. Research groups generally do not share data without some sort of transfer agreement in place that requires the recipient to adhere to certain constraints with regards to publishing and applying for grants. Even if all these conditions are met, the data consumer might simply not have the infrastructure required to store and preserve large datasets.

In the presence of these constraints, it is desirable for a solution that forgoes the need for data to be present locally in order to operate on it. Instead of moving the data to the code, it would be a better idea to move the code to the data itself. An immediate advantage of this scheme is that the controller of the data can put in place various measures to restrict access of their data which can be completely orthogonal to the program logic. These restrictions need only apply at runtime which means that the developer of the software will not need to concern himself with these matters.

This inverse strategy is not a new idea and has been explored in some depth in the existing literature. Existing technologies that achieve this include Remote Procedure Calls (RPCs), CORBA, Mobile Agents and Component Object Models such as EJB. However, these frameworks are rather heavyweight and difficult to retrofit into a large codebase. Incorporating these frameworks in existing applications will probably involve a large amount of effort and might introduce bugs and flaws in the existing code.

Therefore, we seek to develop an easy to use alternative which is as non-invasive as possible. Some of the system level solutions that have been actively developed and researched are process migration and thread migration. Some of the well known systems such as Zap[1] migrate a running process by pausing it, marshalling it and transferring it to another system over the wire, unmarshalling it and restarting it. In order to accomplish the pause/restart, they use a mechanism known as checkpoint-restart. CRAK[2] is a checkpoint-restart system that has been implemented as a kernel module.

However, process migration is a fairly heavyweight operation. For our application, it makes more sense to only migrate a part of the process, that is, migrating a thread of execution. Thread migration has been explored at both the VM level as well as the application level. The former involves changes to the underlying VM while the latter involves aspect oriented programming. In the next section we will discuss some of the approaches to thread migration that have been already explored. The primary reason that we want to explore application level solutions is that invasive changes to the kernel, VM or any other piece of vendor supported infrastructure is anathema if the proposed solution is intended for real world usage. This is because most system administrators will be leery of incorporating patches to existing systems in production that will be unsupported by future releases from the vendor.

In this report, we present two thread migration frameworks; Mobilethreads and Remotelocals, that make thread migration a relatively simple feature to integrate into existing Java programs.

2 Related Work

At the core of any strong migration framework is a mechanism that allows checkpoint/restart (CR) of a process or a thread. The body of work most related to ours is in application specific checkpointing. We now discuss application level checkpointing and userspace CR libraries with a focus on Java. We only consider approaches that do not involve modifications to the JVM and minimal changes to the standard API.

2.1 Program specialization [7]

This approach involves implementing and embedding certain specific interfaces and logic as per some heuristics in user level program. While the approach is sound and complete, this technique requires additional developer effort and assumes that the target source program is available. This is not always acceptable and is impossible to accomplish for non open source programs. While Mobilethreads has been implemented as a library and requires changes to be made to the source, Remotelocals operates only on the bytecode and does not need access to the program source.

2.2 Incremental checkpointing based on code refactoring [8]

This approach involves incorporating logging code that tracks changes to POJOs (Plain Old Java Objects) in the course of program execution. The approach involves a precompilation step where a rule based transformation is applied to the source program. When the application is running, deltas to the program state are logged to the disk and can be referred to when we wish to roll-back. This incremental approach is somewhat different from the snapshot approach of most CR mechanisms. The reason why the authors choose to do it this way is because of the requirements of the primary project; Ptolemy, which is a distributed model execution framework. The implementation of a Time Warp style distribution execution requires the checkpointing facility. Since Ptolemy runs on a large distributed system and each of these components have their own logical state, creating a snapshot of such a setup is not feasible as it involves large scale synchronization.

Their approach discusses various modifications to operations involving side-effects such as variable assignment, array assignments and operations involving collections. For the first two, they introduced a layer of indirection which involves a call to a method that logs and actually performs the side-effect. For the last, they made changes to the Java API.

The evaluation is non-existent and it is our opinion that the logging framework is not very practical at all. Since all assignments need to undergo an extra method call, source level incremental checkpointing could become very expensive.

2.3 Extending Java [9]

This approach involves adding a source level construct to Java that ensures data integrity in the face of incremental checkpointing. The basic checkpoint mechanism is the same as before with added support for exception handling. In the previous work, an exception caused when rolling back or checkpointing would have caused an inconsistent state. In this work, the added source level construct ensures that exceptions are handled in a sane manner.

However, as with the previous work there is no detailed evaluation presented for large programs. Moreover, the addition of unsupported features to the Java compiler is not a practical approach as these features could be invalidated by changes made to the compiler by the vendor. In contrast, both Mobilethreads and Remotelocals work with the standard JDK and Java 1.6 compiler.

2.4 Context sensitive Capture and Replay [10]

The idea here is to use record-replay mechanisms for implementing cr efficiently. Specifically, the work deals with interactions between the program states before and after the checkpoint. Again, this is a source level approach .

The approach employs the use of static analysis to identify certain points in the program where bytecode for checkpointing and bytecode for replaying are inserted. The bytecode responsible for checkpointing captures heap data, values of static variables and object graphs. This mechanism is performed without actually modifying the call stack or the program counter using an approach called backward slicing which essentially removes all code except for the program points. The replaying algorithm restores the relevant captured values at each program point to force the execution to take the correct control flow. owever the work here has some serious limitations. One is that external state is not captured. Thus the states of file descriptors, database connections and socket connections are not preserved, which is essential for process migration.

Second is that this approach is limited only to single threaded programs and does not support multithreaded interleaving. Finally, programs that depend on variables which differ per execution run such as system clock and hash codes cannot be replicated precisely. The record-replay functionality of Remotelocals leverages the Chronicler approach which records multithreaded applications but does not guarantee replay of their interleaving.

2.5 Thread Migration and Checkpointing in Java [11]

This is a RMI based approach for migration and checkpointing. The main contribution here is a rather mature library called PadMig which can perform thread migration and checkpointing in Java. Instead of creating new language constructs, it depends solely on annotations and is thus Java 5 compliant. In order to transmute a non-migratable program to a migratable one, the user merely needs to incorporate PadMig annotaitons at certain specific points, similar to InVivo. This makes the library very easy to use.

For migration to work, the programmer needs to insert migration points into the program. It is at those points that the thread is migrated. Similarly, the programmer may insert checkpoints at those locations, the only difference being that the program execution continues normally. If a method wishes to migrate or call another migratory function, it must be annotated with a PadMig construct. Methods at the bottom of the call stack need to be migrated with a different keyword. As it can be ascertained, these annotations may be misused thus necessitating the need for a verification check at compile time.

When running an application that makes use of PadMig, the environment must also be running the server which intercepts migration requests. PadMig is used in the PubWEB BSP based web computing infrastructure.

PadMig is essentially a source level checkpoint restart system that has been cleverly packaged up using Java annotations. It has a non trivial overhead on the code size and results in significant code bloat when fully compiled. While we have not conducted a thorough analysis of the code bloat resulting from Remotelocals, a preliminary investigation reveals a 20-30% overhead on the code size against nearly 40% with a fully annotated PadMig using application.

3 Mobilethreads

Mobilethreads allows the ability to statefully transfer objects *and the classes on which they are based*, from one application or machine to another, as the application is running.

The destination application or server does not need to contain the classes on which objects it receives are based. In a similar manner, mobile objects can fetch objects from the remote application and return them back to the local system. Only a small server runs on the remote system that receives the objects sent by the Mobilethreads library. These objects are consumed on the remote side and any results are returned to the client once the computations have been completed.

Mobilethreads has been designed to be simple and easy to use, without the need to learn complex APIs, define remote interfaces or to implement the Serializable interface presented by the Java JDK. It is intended to be seamlessly compatible with existing Java code without requiring any significant architectural or code modifications. Additionally, Mobilethreads allows the invocation of arbitrary methods as well as third party libraries without any additional effort.

3.1 Implementation

The implementation of Mobilethreads is relatively straightforward. The user invokes a call to the primary API specifying the IP address of the target machine as well as the resource/object she wishes to transfer to the remote machine. This object is then serialized using a high performance custom serializer, which implements a bytecode level serializer for the Java class. This allows the class being transferred to not have to implement java.io.Serializable.

Once the primary class has been serialized, we also serialize its dependencies and references and send them to the remote machine. This data is cached for the session duration for performance reasons. Mobilethreads implements the serialization, bytecode transfer and caching in a transparent manner using a protocol implemented specifically for this purpose. This protocol is decoupled from the underlying transport protocols and is independent of it. The protocol is stateless and connectionless. The protocol does not require any synchronization or the execution of a start up sequence in order to operate correctly. Moreover it is also asynchronous and nonblocking. Machines at either end can both send any type of message to one another in full-duplex. In order to handle multithreaded applications, Mobilethreads uses UUIDs to multiplex and interleave requests and responses to and from multiple threads on both machines over the same connection. The protocol also supports fire and forget semantics.

Apart from straightforward RPC-style behavior, the protocol supports stateful interactions between applications, despite being stateless itself. When client applications make use of the same session ids, those clients are also able to access the same classes and data stored in static fields in those classes on the remote machines to which they send objects. Therefore, any objects they send will be loaded by the same *session class loader* on the remote machine. Client applications using different session ids, will have their mobile objects loaded by *separate* session class loaders on remote machines, and as such will be isolated from each other. Even if the applications happen to use the same classes, the sessions they access on remote machines will each load their own copy of those classes, and as such data stored in static fields in those classes will not be shared between sessions. This isolation is reminiscent of class loader isolation of deployed applications from each other in most Java web servers and application servers. This constraint however applies to classes loaded from client machines only. Those belonging to the host application and those on its classpath are shared with all sessions.

The Mobilethreads protocol was designed with high performance and scalability in mind. It has currently been tested with the geWorkbench tool and performs well in its environment.

3.2 TCP Implementation

It may be observed that establishing a single multiplexed TCP connection between a pair of machines is likely to be more efficient than establishing many connections, however there are still some valid reasons to establish multiple TCP connections. One reason could be a thread-to-thread connection which maps objects from one thread to another on a remote machine in a one-to-one manner.

The implementation of the protocol atop TCP is therefore as follows. The library establishes a single TCP connection to a destination, and multiplexes requests from all threads via this connection by default. Applications are allowed to send requests via additional *auxiliary* connections. The library then routes responses via the same connection from which it receives a request, effectively allowing applications to open multiple streams to a destination when, and only when necessary.

3.3 Comparison with conventional technologies

Conventional RPC is generally implemented using remote interfaces and proxy objects; also known as stubs. The server application typically defines an interface containing the signature of a method (or “procedure”) that it wishes to make available over the network. Internally, the server contains a class which implements the methods of this interface. The server requests the RPC framework server-side to “bind” its implementation of the interface to a particular port on the server, or to a particular name in a directory-like structure.

The client application also contains a copy of the interface. The client application requests the RPC framework client-side to connect to, or “look up” an implementation of the interface on the specified remote server. The RPC framework creates a proxy object (or “stub”) implementation of the interface client-side, which when invoked relays method calls across the network to the real implementation on the server.

Mobilethreads takes a completely different approach. Rather than specify which methods must be made available over the network in advance, the library allows the client application to transfer a regular Java object from the local JVM into the application in the remote JVM. Once in the remote JVM, the object is free to call any methods in the remote application, and if it wishes, return objects from the remote JVM back to the client application. This is essentially, weak thread migration, where the thread-like object is migrated to the remote machine but the context of execution is not.

This aspect of the library comes into play because it does not require the classes which implement the object to exist in advance in the remote application. The library automatically loads classes from the client application into the remote application and caches them there for the duration of the session. Sessions can be short-lived (ad-hoc), long-lived, or persistent. Given that objects can be transferred back-and-forth statefully between machines, and interact with remote applications, it becomes possible to implement much more sophisticated applications and design patterns than RPC alone would allow. This allows the possibility of developing newer, more sophisticated design patterns as well as architectures for distributed applications.

3.4 Evaluation of Mobilethreads

We used Mobilethreads with geWorkbench and were able to execute code on remote data. One of the interesting aspects of using Mobilethreads with geWorkbench is that the developer has to be careful with the object that she is intending on serializing and sending over. This is an important issue because since the serialization process will attempt to marshall all references held by the object, if that object holds a reference to the current thread then it will by transitivity, pull in all objects associated with the current thread group. This typically results in a buffer overflow on the test machine. Normally, the classes whose objects are to be sent over should be defined as separate compilation units or they should be static nested classes within the original class, so that they do not hold any references to any objects in the current context. Apart from this slight snag, using Mobilethreads simply involved executing a single line of code. We forsee that if one uses IoC containers to initialize the remoting object, all configuration and initialization can be accomplished via an XML file. This is left for future research.

4 Remotelocals

Migrating the execution context of a thread typically requires VM support although it is possible to achieve the same at the application level with bytecode manipulation (aspect oriented programming). When migrating a Java application, we need to transfer all the components of its state. In Java land, we have that

object = <program state (bytecode), data state, execution state>

execution state = <program counter (thread unique), java stack>

java stack = [frames], frame = <locals, operand stack>

The operand stack is the stack we normally manipulate.

Strong/transparent migration involves suspending execution of a thread, capturing its execution state and serializing it while finally sending it to a target location and re-instating the execution state. If this is done in a general manner, it is strong. If the programmer is expected to intervene, it is not.

The approach used by Remotelocals is as follows. We use a distributed master-master replicated database for distributed memory sharing. This allows us to ensure that changes made by a thread in a local machine will be visible to threads on remote machines. The class files are instrumented with recording as well as replaying code and field variables are instrumented so that when we reconstruct the state, their values are fetched from the the replicated database. Remotelocals uses Mongodb, a fast nosql database for memory sharing. Master-master replication is achieved using “mmm”, a Python application that guarantees eventual consistency in a replicated MongoDB cluster.

4.1 Implementation

The first step that we need to accomplish is to setup the replicated server for distributed memory sharing. We note that all the heap data can be referenced by the field variables in the thread as well as the global data accessible to it. We instrument the bytecode of the thread object such that it acquires new getters/setters for each of its fields that allow it to fetch and commit content from the replicated server. The constructor is instrumented so that the connection to the replicated server is established before the thread starts to execute. We replace all instances of the GETFIELD and PUTFIELD bytecode to calls to the respective getters/setters.

Remotelocals implements thread migration in a non-preemptive manner. That is, migration only occurs at programmer designated locations. Two primitives that we need to use are capture() and resume(), methods called when we need to migrate and when we need to resume execution, respectively. Both of these primitives must be called from within the context of the thread being executed. So if a thread wants to migrate itself, it calls capture(). When, we wish to resume, the thread controlling the resumption of the paused thread calls resume() on the migrating thread after the migration has completed. In order to enable state capture, after each INVOKEX bytecode instruction, we insert a capturing routine that is executed only if a flag is set. That flag will be set in the capture() method. This routine inserts a return insn to suspend execution of the method. For each method on the stack, we save the stack frame before the INVOKEX insn. The index of the INVOKEX insn is the program counter. Once we hit the return insn, we return from the current method and move on to the next method in the callstack, which has also been similarly instrumented. This goes on recursively until we reach the end of the call stack and have captured the execution state of the thread.

Each method has a designated method frame whose fields represent the local variables of a method. Note that we do not use arrays as the performance cost of the AASTORE and AALOAD bytecode is higher than a GETFIELD and PUTFIELD. When saving the state of a method, we store its local variables in a newly initialized method frame. The frames themselves are stored in a global array that tracks the frames associated with the current call stack.

In order to minimize the logging overhead of state capture, we perform a MOD analysis to determine the variables that are going to be modified in a method. If the variables are merely used as references, then we do not log them as their values can be directly fetched from the replicated server. This results in a reduced logging overhead. Apart from a MOD analysis, we also allow sources of nondeterminism to be logged for replay. This mirrors the Chronicler approach [15], where the authors have shown that logging nondeterminism is lightweight and complete. In order to designate a method as a record-target instead of a state-capture target, we simply annotate it as such. While recording nondeterminism leads to a lighter logging process, replay can possibly take a long time. This tradeoff involving lighter logging process vs longer replay is a topic we have left for future research. In order to pause we insert artificial returns in the code that allows us to exit the context of the thread in a safe manner.

In order to restore the execution state we need to invoke all the relevant methods in the same order as they were captured. This is done by inserting a state restoring code block in the beginning of the method which consists of a switch instruction. The switch instruction reads the program counter we had stored and jumps to the appropriate case which restores the state as well as jumps to the appropriate INVOKEX insn. This step requires the insertion of arbitrary GOTOs in the bytecode which is a tricky endeavor. The library we used to perform bytecode manipulations inserted random IMPDEP instructions in the instrumented code. This bytecode is a restricted operation to be used only by the debugger, if the verifier observes this prior to startup it will throw a verification error. We had to manually replace those instructions as a post processing step. We eventually end up in capture() a second time through which we can set the capturing flag to false. After the flag has been set to false, we continue the recording process in anticipation of another migration event.

4.2 Comparison with existing technologies

The core methodology of thread migration at the bytecode level has not changed much. Apache JavaFlow is a well known library that allows continuations in Java and makes use of concepts involving thread migration. These concepts were first enunciated in the Brakes system. Its methodology was significantly improved upon by Kilim, an actor framework for Java. Kilim allows a programmer to multiplex thousands of contexts of execution onto a single Java thread. This involves starting and stopping those contexts of execution. Kilim makes use of all the optimizations we have described save a few. Kilim performs live variable analysis in order to minimize the logging process. Remotelocals on the other hand can afford to use the MOD analysis, because if the value of a variable is unchanged, during replay we can simply fetch it from the MongoDB server. Secondly, we leverage record-replay for the start-stop mechanism unlike Kilim or Brakes. This results in a lighter logging process albeit a greater replay time.

4.3 Evaluation of Remotelocals

No significant evaluation of Remotelocals has been done yet although this is planned for future research. Since Remotelocals builds on top of Kilim's optimizations, its performance is atleast as good as Kilim's. However, we foresee that that the additional MOD analysis will lead to greater payoff when experimenting with real world workloads such as Dacapo. Additionally, we plan on implementing several design patterns using the functionality offered by Remotelocals and explore their impact on the way Java applications are designed and written.

5 Conclusions

In this report we presented two systems for application level thread migration. Mobilethreads is a library that allows weak thread migration to be deployed in existing Java applications without much effort from the developer. Its API requires only the IP address of the target machine and the Java object that needs to be sent over. This object is serialized and transferred to the remote machine over the wire using protocol buffers, where it is deserialized and executed. The remote machine needs to have a server running that can intercept incoming objects, no other demands are made on it. As an optimization, all the dependencies and requirements of the Java object sent over are cached on the remote server for the session duration. The protocol used to transfer objects and receive results has been implemented on top of TCP but is agnostic of the transport layer. This protocol is fast, scalable and simple to reason about. Mobilethreads has been deployed in the geWorkbench suite of applications.

Remotelocals is a system that makes application level strong thread migration possible in Java using bytecode manipulation. It makes use of Mongodb to share memory state and relies on instrumentation inserted at the bytecode level in order to save and restore state. It uses a mixture of efficient state capture and record-replay techniques in order to achieve a low logging overhead. Using these two systems, it is very easy to achieve thread migration of desired strength in existing Java applications.

6 Bibliography

[1] ZAP: Transparent Checkpoint-Restart and Migration Using Operating System Virtualization, Nieh et al , Proceedings of Usenix OSDI 2002

[2] CRAK: Linux Checkpoint/Restart As a Kernel Module, Zhong et al, Techreport CUCS-014-01

[3] Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters

[4] DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop,

[5] Linux cr lwn article: https://ckpt.wiki.kernel.org/index.php/Main_Page

[6] CR survey: http://lwn.net/Articles/412749/

[7] Incremental Checkpointing based on Java Source Code Refactoring, TH Feng

[8] Julia L. Lawall and Gilles Muller. 2000. Efficient Incremental Checkpointing of Java Programs. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*(DSN '00). IEEE Computer Society, Washington, DC, USA, 61-70.

[9] Extending Java with checkpointing, TH Feng

[10] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. 2007. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*(ESEC-FSE '07). ACM, New York, NY, USA, 85-94. DOI=10.1145/1287624.1287638 http://doi.acm.org/10.1145/1287624.1287638

[11] Thread Migration and Checkpointing in Java, T. Gehweiler

[12] Chronicler: Lightweight recording for reproducing field errors, Bell et al., ICSE 2013 (To appear)